Разработка и анализ алгоритмов

Лекция 12
Красно-чёрное дерево. Дерево отрезков

Сергей Леонидович Бабичев

2-3-4 деревья.

2-3-4 деревья

- AVL-дерево — прекрасная структура данных.
- Её немного портят две вещи:
  - различная высота узлов;
  - необходимость постоянной балансировки.
- Как мы уже выяснили, в бинарных деревьях приходится чем-то жертвовать.
- В B-деревьях высота всех поддеревьев одинакова, но процедуры вставки/удаления трудоёмки и B-деревья больше рассчитаны на внешнюю память.
- Давайте объединим двоичные и B-деревья в 2-3-4 деревьях.

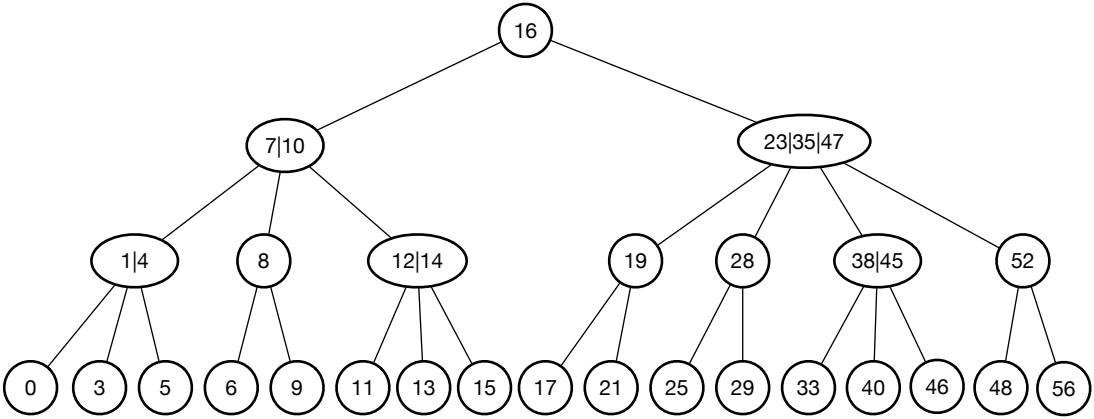
2-3-4 деревья

Definition (2-3-4 дерево)

2-3-4 дерево поиска — дерево поиска, которое либо пусто, либо в котором содержатся узлы трёх типов — 2-узлы, 3-узлы и 4-узлы. 2-узлы содержат один ключ k_1 и две ссылки на элементы $k < k_1$ и элементы $k \geqslant k_1$. 3-узлы содержат два ключа k_1 и k_2, $k_1 < k_2$. и три ссылки. Первая ссылка — на элементы $k < k_1$. Вторая — на элементы $k_1 \leqslant k < k_2$ и третья — на элементы с $k \geqslant k_2$. 4-узлы — содержат, соответственно, три ключа $k_1 < k_2 < k_3$ и четыре ссылки.

Definition

Сбалансированное **2-3-4 дерево** — 2-3-4 дерево поиска, в котором все пустые находятся на одной высоте.


16
7|10
23|35|47
1|4
8
12|14
19
28
38|45
52
0
3
5
6
9
11
13
15
17
21
25
29
33
40
46
48
56

Поиск в 2-3-4 дереве

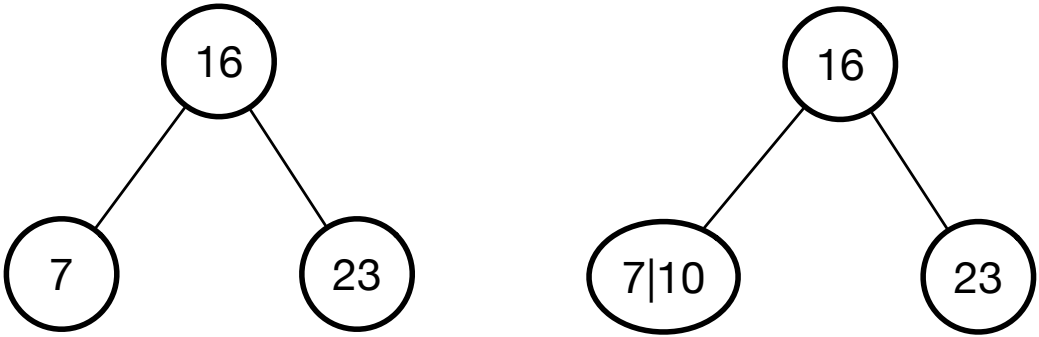
1. Поиск начинается в корне.
2. Если ключ равен хотя бы одному ключу в узле — возвращаем узел.
3. При выполнении нужного условия в узле — рекурсивно вызываем себя для нужного подузла.

Вставка в 2-3-4 дерево

- Классическая вставка в BST-дерево (поиск и вставка в терминальный узел) плоха — дерево теряет баланс.
- Но это — 2-3-4 дерево и баланс можно соблюсти.
- Если узел для вставки — 2-узел, то превращаем его в 3-узел.
- Если узел для вставки — 3-узел, то превращаем его в 4-узел.
- Вставка в 4-узел требует разбиения узла на два 2-узла.
- Если в родительском узле есть место (это 2- или 3-узел), то операция успешна.
- А что делать, если родитель — 4-узел?
- Опять разбить его и передать вверх?
- Нужно обеспечить, чтобы при спуске вниз 4-узлы не могли бы быть концом поиска.

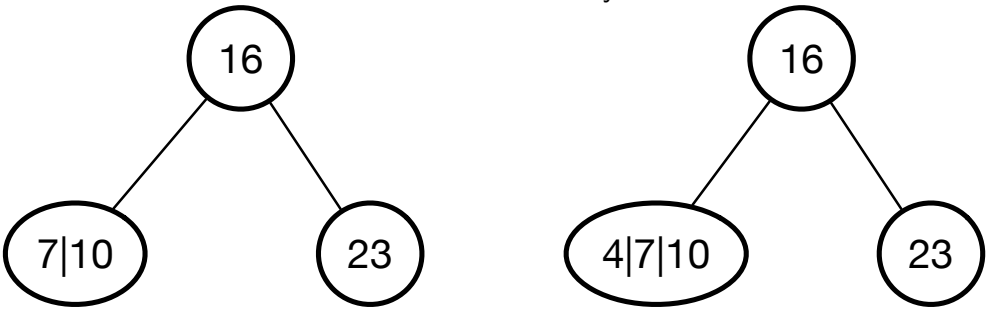
Вставка в 2-узел

Вставляем ключ 10 в 2-узел.



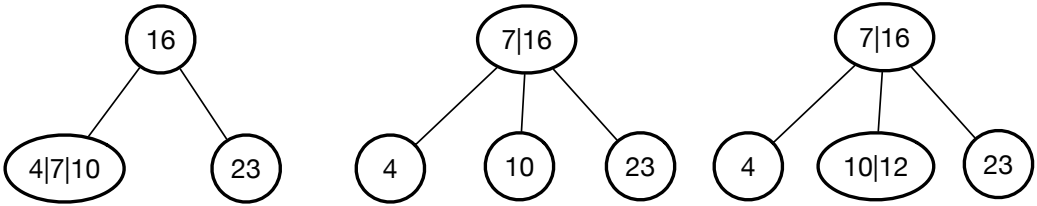
Вставка в 3-узел

Вставляем ключ 17 в 3-узел.



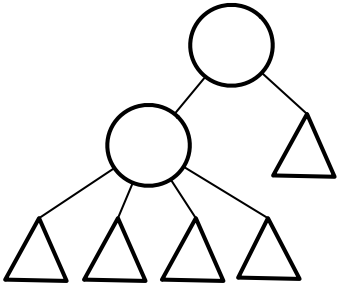
Вставка в 4-узел

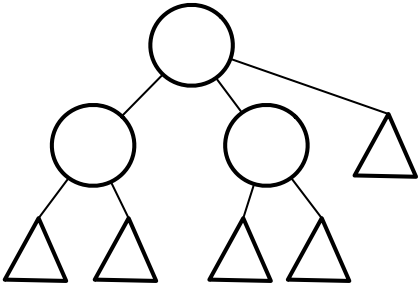
Вставляем ключ 12 в 4-узел. Разбиваем 4-узел на два 2-узла (4) и (10). Вставляем средний узел (7) в родительский 2-узел. Повторяем вставку (12) уже в 2-узел.



Обеспечение отсутствия 4-узлов в месте вставки.

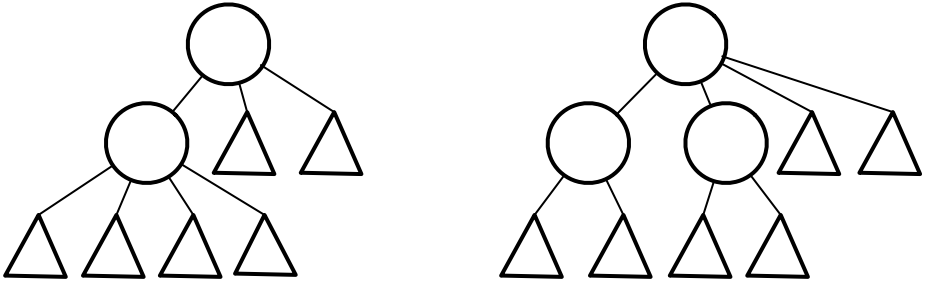
- Наблюдение: при перемещении ключей можно перемещать и ссылки.
- Варианты появления 4-узла при вставке:
  1. 2-узел с дочерним 4-узлом. Разбиваем 4-узел на два 2-узла. Родителя делаем 3-узлом.





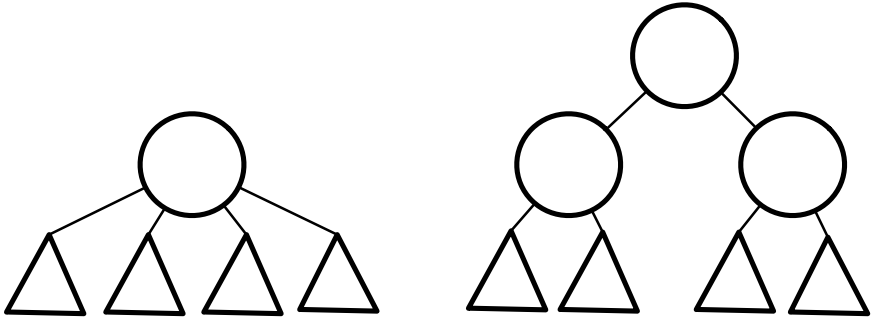
Обеспечение отсутствия 4-узлов в месте вставки.

- Варианты появления 4-узла при вставке:
  2. 3-узел с дочерним 4-узлом. Разбиваем 4-узел на два 2-узла. Родителя делаем 4-узлом с одной новой ссылкой.



Увеличение высоты дерева

- Корень может отказаться 4-узлом.
- Разбиваем его на группу из трёх 2-узлов.



- Это — единственный случай, когда увеличивается высота дерева. Одновременно для всех узлов.

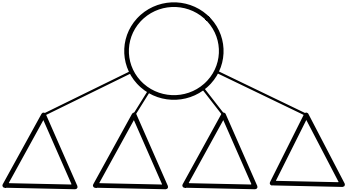
Идеальное дерево?

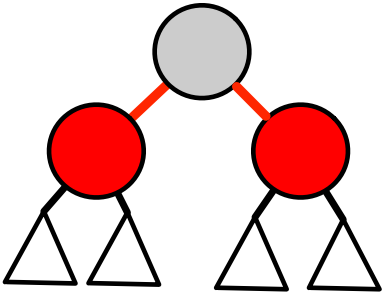
- Данное дерево абсолютно сбалансированно.
- Операции вставки увеличивают высоту дерева, требуя в редко встречающемся худшем случае $\log N$ преобразований.
- Эмпирические исследования показали, что количество разбиений близко к $O(1)$.
- Самый большой недостаток — нужно работать с тремя типами узлов.
- Поменяем представление на привычное — 2-деревья.

Преобразование 2-3-4 дерева в 2-дерево

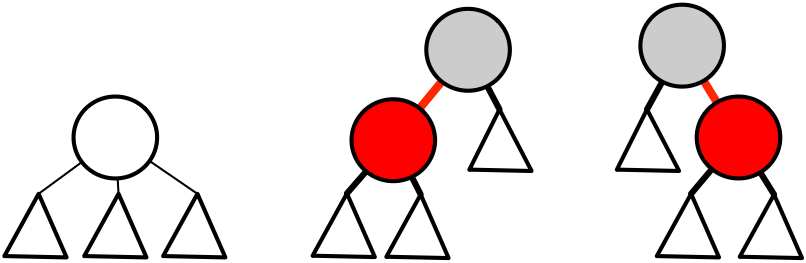
- Попробуем представлять 2-3-4 деревья в виде стандартных BST-деревьев, с некоторым добавлением.
- Каждый 3- и 4-узел будет неким блоком (фиксированным куском) 2-узлов.
- 4-узел имеет фиксированную форму.
- 3-узел имеет два варианта.
- Связи, цементирующие 3- и 4- узлы пометим красным цветом.
- Остальные связи — чёрным цветом.
- Цвет узла определяем по цвету пришедшей связи.

Структура 4-узла





Структура 3-узла



Красно-чёрные деревья

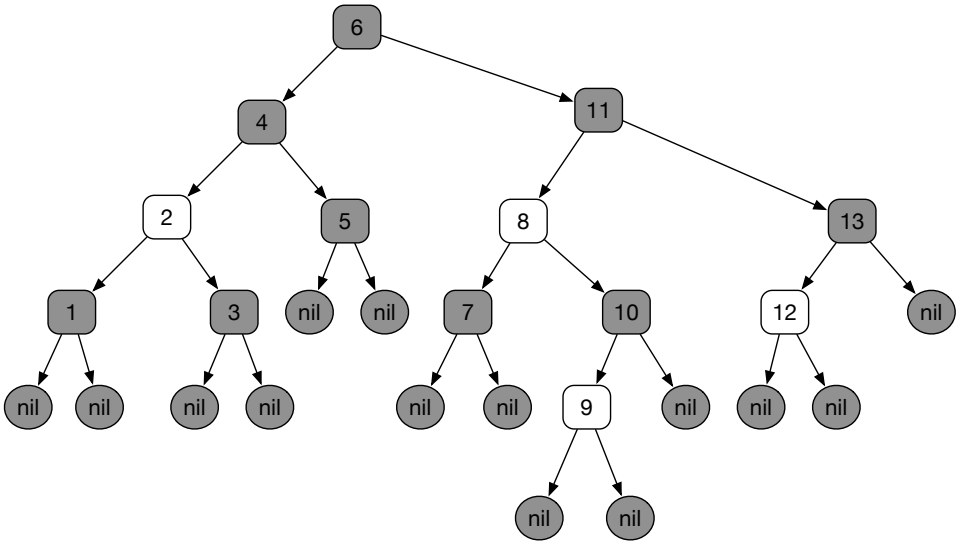
- Красно-чёрные деревья (RB) — представление в виде 2-деревьев 2-3-4 деревьев.
- Все соображения о 2-3-4 деревьях справедливы для RB.
- Строительные блоки RB-дерева — 3-узлы и 4-узлы 2-3-4 дерева.
- RB-деревья сочетают простоту поиска BST-дерева с относительной простотой вставки и балансировки 2-3-4 дерева.
- Затраты на организацию невелики: 1 бит на цвет и поле родителя.
- Основная проверка: если оба ребёнка красные, то узел — часть 4-узла.
- Основные операции можно производить малыми поворотами и перекрашиванием.

Красно-чёрные деревья: общее определение и свойства

Красно-чёрное дерево это сбалансированное бинарное дерево поиска.

1. Вершины разделены на **красные** и **чёрные**.
2. Каждая вершина имеет указатели *left*, *right*, *parent*.
3. Каждая терминальная вершина (лист) — чёрная.
4. Если вершина — красная, то её дети — чёрные.
5. Все пути от корня $root$ к листьям содержат одинаковое число чёрных вершин. Это число называется **чёрной высотой дерева**, **black height**, $bh(root)$

Красно-чёрные деревья



Красно-чёрные деревья

Theorem

Красно-чёрное дерево с N внутренними листьями имеет высоту не более $\log_2(N+1)$

Доказательство.

- Для листьев чёрная высота равна нулю.
- Докажем, что $|T_x| >= 2^{bh(x)}$.
- База индукции: Пусть вершина x является листом. Тогда $bh(x) = 0$ и $|T(x)| = 0 < 2^{bh(x)}$
- Пусть вершина x не является листом и $bh(x) = k$. Тогда для обоих потомков $bh(l) \geqslant k-1$, $bh(r) \geqslant k-1$, т. к. красный будет иметь высоту k, чёрный — $k-1$.
- По предположению индукции $|T_l|, |T_r| >= 2^{k-1} \to |T_k| = |T_l| + |T_r| >= 2^k - 1$ □
- По свойству (4) не менее половины узлов составляют чёрные вершины.
- $bh(t) \geqslant H/2$
- $N \geqslant 2^{H/2} - 1$

$$H \leqslant 2 \cdot \log_2 N + 1$$

□

Красно-чёрные деревья: технические детали

- Удобно использовать не `NULL` и не `nullptr`, а для каждого дерева иметь свой чёрный узел nil.
- Разбиваем структуру данных на две части: интерфейсную и реализационную.
- При обычной вставке свойства красно-чёрности могут нарушаться.
- Для изменения красно-чёрности применяется корректировка цветности — fixup.
- fixup сводит структуру дерева к комбинации 2-узлов, 3-узлов и 4-узлов.
- Вспомним: при вставке необходимо обеспечить отсутствие 4-узла в месте вставки.
- Все операции с 2-3-4 узлами можно проводить обычными поворотами.
- Красные вершины определяют красные цементирующие связи в 2-3-4 дереве.

Красно-чёрные деревья: структуры данных

```
typedef int T;

enum color_e {
    RED, BLACK
};
typedef struct rbnode_s {
    struct rbnode_s *left, *right, *parent;
    enum color_e  color;
    T      key;
} rbnode;

typedef struct rbtree_s {
    rbnode *root;
    rbnode *nil;
} rbtree;
```

Поиск в RBT

- Поиск в RBT ничем не отличается от поиска в любом BST.

Повороты в RBT

- Имеют следующие отличия от поворотов в BST:
  1. Всегда имеется родитель. Поворот должен корректировать и родителя.
  2. Отдельно имеется корень дерева. Если операции задевают корень, он должен быть изменён.
  3. После вращения не требуется возвращать новый корень, структура дерева достижима из отдельного корня и родителей.

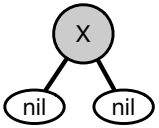
Вращение узла в RBT

```
static void rotateRight(rbtree *r, rbnode* head) {
    rbnode *new_head = head->left;
    head->left = new_head->right;
    if (new_head->right != r->nil) {
        new_head->right->parent = head;
    }
    new_head->parent = head->parent;
    if (head->parent == r->nil) {
        r->root = new_head;
    } else {
        if (head == head->parent->right) {
            head->parent->right = new_head;
        } else {
            head->parent->left = new_head;
        }
    }
    new_head->right = head;
    if (head != r->nil)
        head->parent = new_head;
}
```

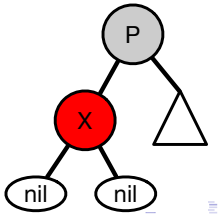
Красно-чёрные деревья: вставка

- Важное замечание: после всех операций корень должен оставаться чёрным.
- Рассмотрим варианты, когда родитель ребёнка слева от дедушки. Для правого случая всё симметрично.
- Вставляем почти как в обычное бинарное дерево поиска.
- Доходим до терминального узла.
- Красим узел в красный цвет, дети — NIL, т. е. чёрные.
- Далее — варианты:

1. Это — единственный элемент. Красим в чёрный цвет.



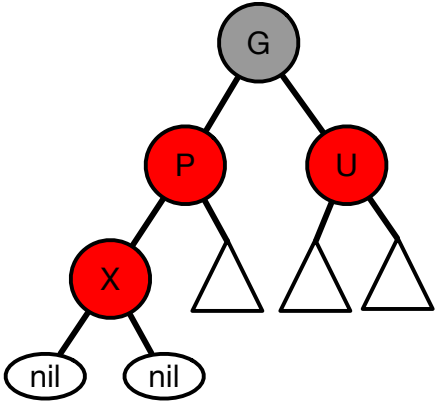
2. Родитель — чёрный. Инварианты не поменялись.



Красно-чёрные деревья: вставка

3. Родитель — красный. Коллизия. Вопрос: красный ли дядя?

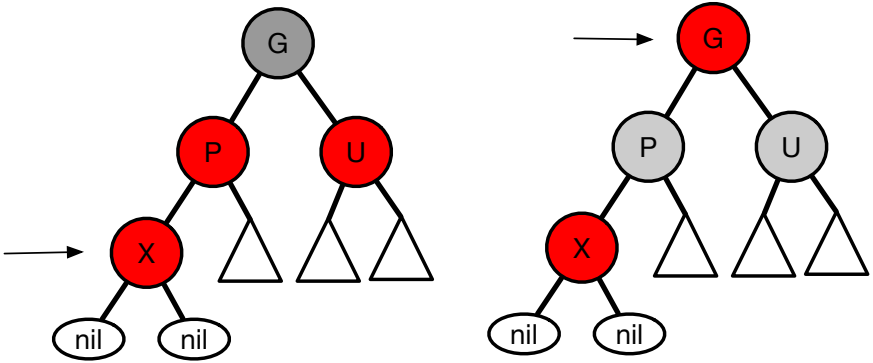
3.1 Дядя — красный. Это просто.



Красно-чёрные деревья: вставка

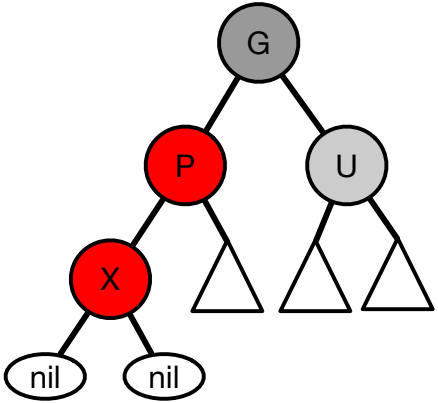
3. Родитель — красный. Коллизия. Вопрос: красный ли дядя?

3.1 Дядя — красный. Это просто. Перекрашиваем родителя с дядей в чёрный цвет и дедушку в красный. Поднимаемся до дедушки и повторяем алгоритм.



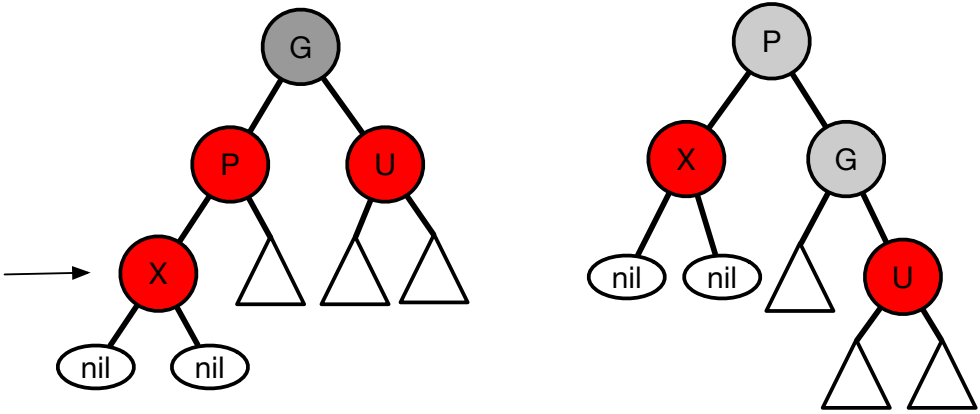
Красно-чёрные деревья: вставка

3.2.1 Дядя — чёрный. Перекраской не отделаешься. Придётся вращать. Может случиться малый поворот, если X — левый ребёнок.



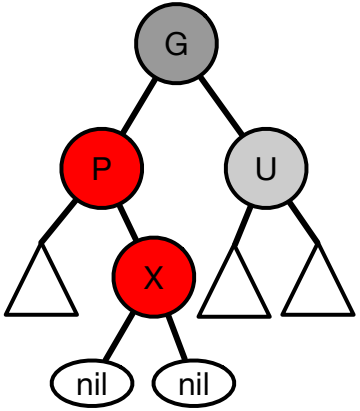
Красно-чёрные деревья: вставка

3.2.1 Дядя — чёрный. Одной перекраской не отделаешься. Придётся вращать в дедушке. Может случиться малый поворот, если X — левый ребёнок. Так как в зоне изменения корень остался чёрным — операция закончена.



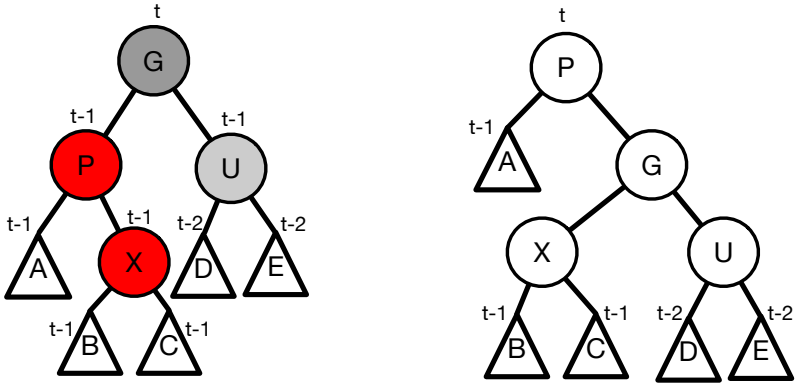
Красно-чёрные деревья: вставка

3.2.2 Дядя всё ещё чёрный, но X — правый ребёнок.



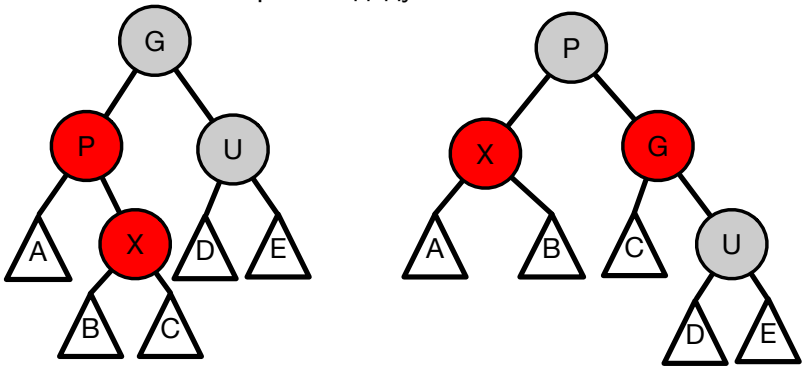
Красно-чёрные деревья: вставка

3.2.2 Дядя всё ещё чёрный, но X — правый ребёнок. Докажем, что правый поворот в дедушке невозможен. Пусть чёрная высота оригинальном фрагменте была t. Тогда чёрные высоты фрагментов A, B, C равны $t-1$, а D и E — $t-2$. После вращения чёрная высота не должна измениться. Тогда P — чёрная, а G и X — красные, что противоречит свойствам: нет двух красных узлов подряд.



Красно-чёрные деревья: вставка

3.2.2 Поэтому, как и в AVL-дереве, нужно предварительно повернуть влево в родителе, а затем — вправо в дедушке.



```
void rb_insert_fixup(rbtree *r, rbnode *x) {
    if (x == r->root) {
        x->color = BLACK; return;
    }
    while (x->parent->color == RED) {
        if (x->parent == x->parent->parent->left) {
            rbnode *uncle = x->parent->parent->right;
            if (uncle->color == RED) {
                x->parent->color = BLACK;
                uncle->color = BLACK;
                x->parent->parent->color = RED;
                x = x->parent->parent;
            } else {
                if (x == x->parent->right) {
                    x = x->parent;
                    rotateLeft(r, x);
                }
                x->parent->color = BLACK;
                x->parent->parent->color = RED;
                rotateRight(r, x->parent->parent);
            }
        } else {
```

Красно-чёрные деревья: удаление

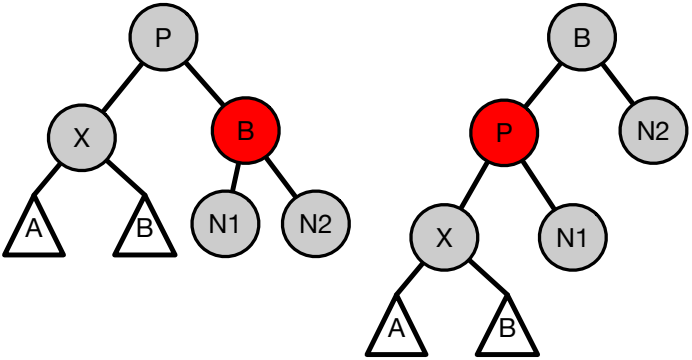
- Начало удаления — классика.
- Узел — листовой. Указатель родителя на него ставим в NIL.
- У узла один ребёнок. Поднимаем его на место удалённой.
- У узла — два ребёнка. Находим минимальный элемент в правом поддереве. У него нет двоих детей, удаляем его перенеся ключ в наш узел.

Красно-чёрные деревья: жизнь после удаления — этап 1.

- Саму вершину мы удалили, у неё либо не было детей, либо был ровно один ребёнок, которого обозначим X.

4. Если мы удаляли красную вершину или пришли в корень, то всё уже сделано.

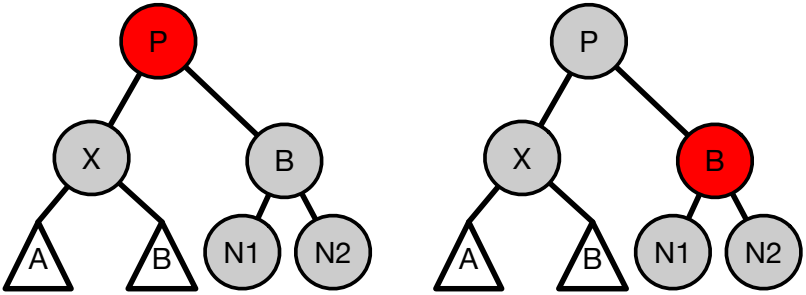
5.1. X — чёрный. Нас заинтересовал цвет брата X. Нам нужно сделать его чёрным для следующего этапа. Если он красный — поворачиваем узел отца влево (вращение с братом) и перекрашиваем их: отца в чёрный, брата в красный. Теперь у X — чёрный брат $N1$, что нам и надо.



Красно-чёрные деревья: жизнь после удаления — этап 2.

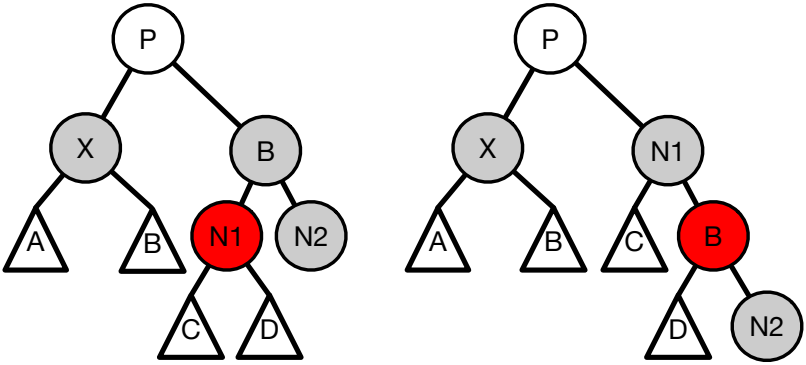
5.2. Здесь X — чёрный, родитель красный и брат чёрный. Смотрим на племянников.

5.2.1. Если оба племянника чёрные. Достаточно перекрасить родителя в чёрный и брата в красный, переходим в родителя и повторяем алгоритм с пункта 4.



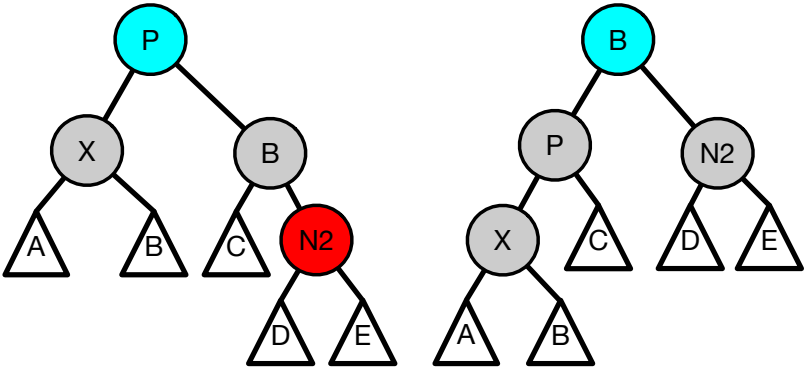
Красно-чёрные деревья: жизнь после удаления — этап 3.

5.2.2. Если правый племянник чёрный, левый — красный. Перекрашиваем брата в красный цвет, левого племянника в чёрный и повернём их. Перевычислим брата и перейдём к последнему этапу.



Красно-чёрные деревья: жизнь после удаления — этап 4.

5.2.2. Если правый племянник красный, то красим брата в цвет родителя, красного племянника в чёрный и вращаем родителя влево. А вот теперь — всё.



Красно-чёрные деревья vs АВЛ-деревья

	RB-tree	AVL-tree
Средняя высота	до 1.38H	H
Поиск/вставка	до 1.38t	t
Поворотов при вставке	до 2	до 1 большого
Поворотов при удалении	до 3	до $\log N$
Дополнительная память	1 бит + parent	1 счётчик

Дерево отрезков

Дерево отрезков

Пусть нам надо решить задачи:

- Многократное нахождение максимального значения на отрезках массива.
- Многократное нахождение суммы на отрезке массива.

Мы умеем совершать эти действия за время $O(N)$, где $N = R - L + 1$. При определённой подготовке их можно совершать за $O(\log N)$.

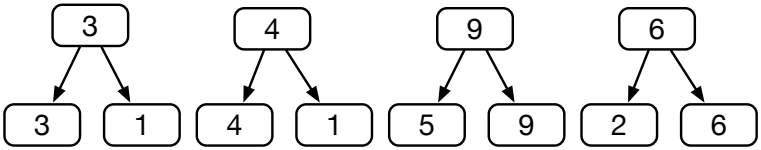
Дерево отрезков

Попробуем воспользоваться бинарными деревьями.
Для примера возьмём массив {3, 1, 4, 1, 5, 9, 2, 6}.

Вот как выглядит этот массив:

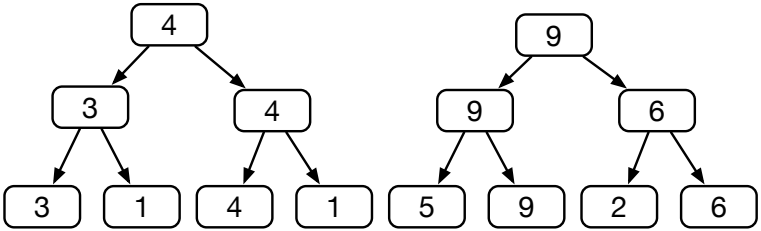
Дерево отрезков

Попарно соединим соседние вершины, поместив в узел-родитель значение функции `max(left,right)`.



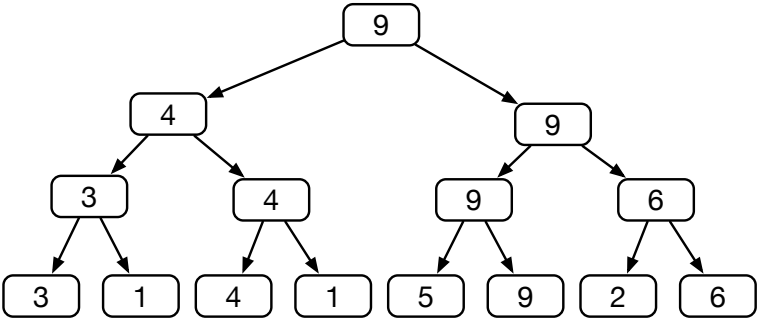
Дерево отрезков

Проделаем эту же операцию от получившихся узлов:



Дерево отрезков

Наконец:



Родитель каждого узла называется *доминирующим узлом*.

Дерево отрезков: представление

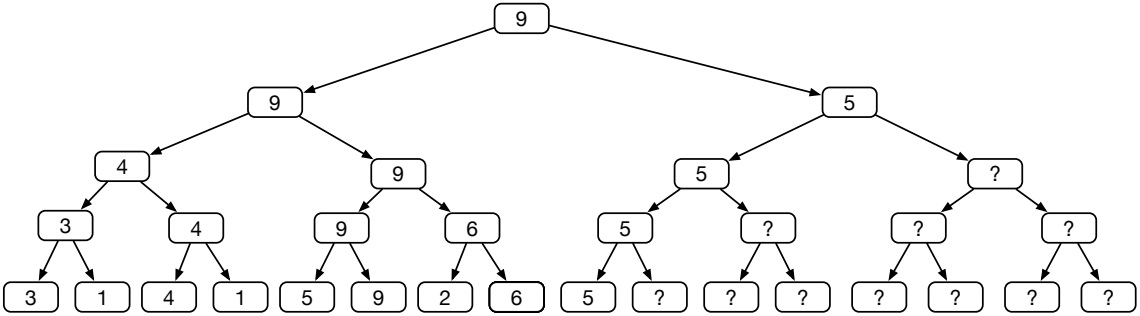
Возможный вариант представления — обычное бинарное дерево с указателями.

- На каждый узел требуется два указателя вниз.
- Для удобной работы требуется индикатор «левый/правый узел» и один указатель на родителя.
- Минимум 4 элемента на узел.

Но ведь это же полное бинарное дерево? Тогда почему не использовать бинарную кучу?

Дерево отрезков: бинарная куча

Бинарная куча требует полного бинарного дерева. Количество элементов должно быть степенью двойки.



Что должно находиться в узлах, отмеченными знаками вопроса?

Дерево отрезков

Все значения в узлах вычисляются с помощью функции

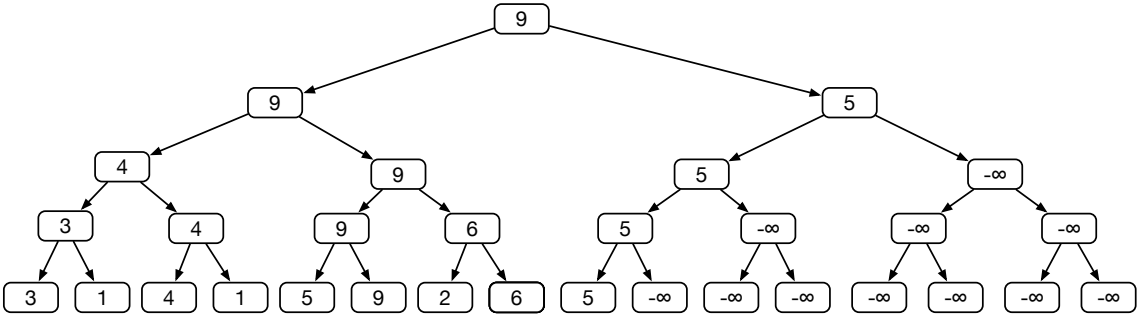
$$P = \max(L, R).$$

Чтобы не плодить сущности, то же самое должно происходить с элементом `'?'`.

То есть элемент `'?'` есть $-\infty$.

Для функции `max` число $-\infty$ есть *нейтральный элемент*.

Дерево отрезков



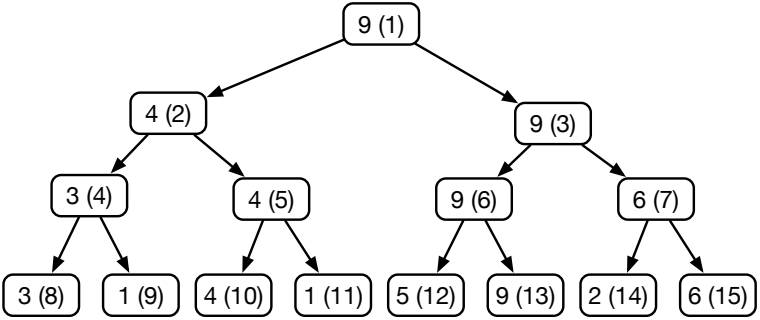
Дерево отрезков

Идея дерева отрезков распространяется на все такие функции, в которых:

$$\begin{aligned} A \circ B &= B \circ A \\ A \circ (B \circ C) &= (A \circ B) \circ C \\ \exists E : A \circ E &= A \end{aligned}$$

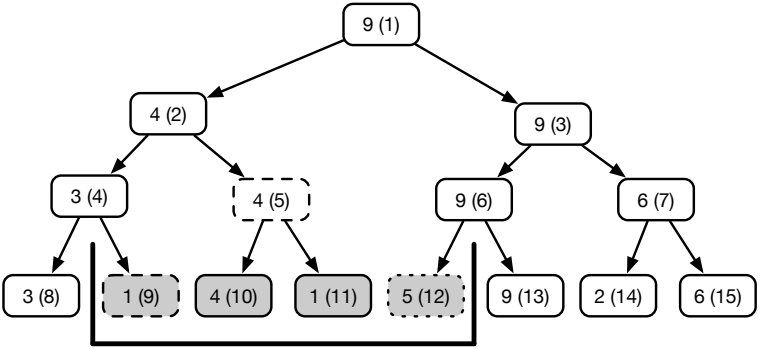
Операция	Нейтральный элемент
`max`	$-\infty$
`min`	$+\infty$
`+`	0
`*`	1

Дерево отрезков: алгоритмы



- ***Create(size):*** создаётся полное бинарное дерево, инициализированная нейтральными элементами. $C = \min(2^k) : C \geqslant size$.
- ***Insert/Replace(i, val):*** `body[i+C]=val; propagate(i);`

Дерево отрезков: функция на отрезке



- ***Func(left,right):***
  - `Res = E`
  - `if (left % 2 == 1) Res = Op(Res,body[left++])`
  - `if (right % 2 == 0) Res = Op(Res,body[right--])`
  - `if (right > left) Res = Op(Res,Func(left/2, right/2))`

Дерево отрезков

Сложность операций:

- Требуемая память: $\min = O(2N) \ldots \max = O(4N)$.
- Операция ***Insert/Replace***: $O(\log N)$.
- Операция ***Func*** на любом подотрезке: $O(\log N)$.